Grid Computing(10/29)

12M37037 Iwabuchi Keita

A GPU implementation of inclusion-based points-to analysis

Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali.
(PPoPP'12)

1. Introduction

1.1 GPU Computing

- GPU
  - GPU hardware is designed to process blocks of pixels at high speed and with wide parallelism
  - well suited for executing regular algorithms that operate on dense vectors and matrices
- Irregular algorithm
  - Irregular algorithm use dynamic data structure(Graph,Tree)
  - BFS, n-body simulations etc.

1.2 Graph algorithm on GPU

- Most of Irregular algorithms that have implemented on GPU do not modify the structure of graph
  - modifications can be predicted statically and appropriate data structures can be pre-allocated for the program
- morph algorithm
  - edges or nodes are dynamically added to (or removed from)
  - compiler optimizations, social network maintenance
  - Implementation of a morph algorithm on a GPU is challenging
    - how to support dynamically changing graph on a GPU

1.3 contributions

- A GPU implementation of Andersen's points-to analysis
    - useful for understanding some of the differences between optimizing codes for multicores and GPUs
- Propose Graph data structure(of morph algorithms) suited for GPU
    - allowing to add and remove edges dynamically
    - takes into account three performance factor
      global address alignment, shared memory bank conflicts and thread divergence.
- GPU code outperforms an existing CPU version
    - achieving an average speedup of 7x

2 Inclusion-based points-to analysis

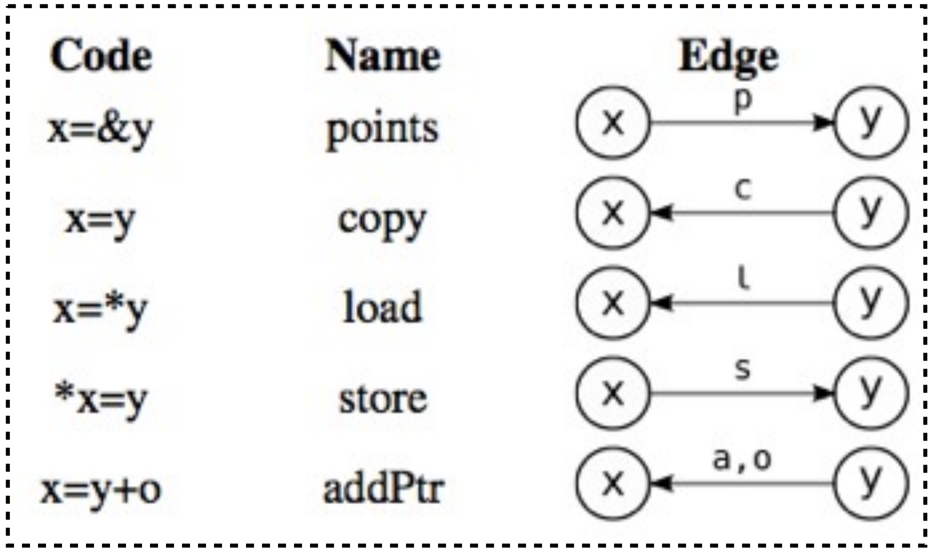
2.1 Andersen-style points-to analysis

- **Points-to analysis algorithm**
- **A popular algorithm (also called inclusion-based analysis)**
- **The asymptotic worst-case complexity is O(n^3) ,where *n* is the number of variables**
- **Can be formulated in terms of graph rewriting rules**

Step 1,2 : Initialization, Graph creation

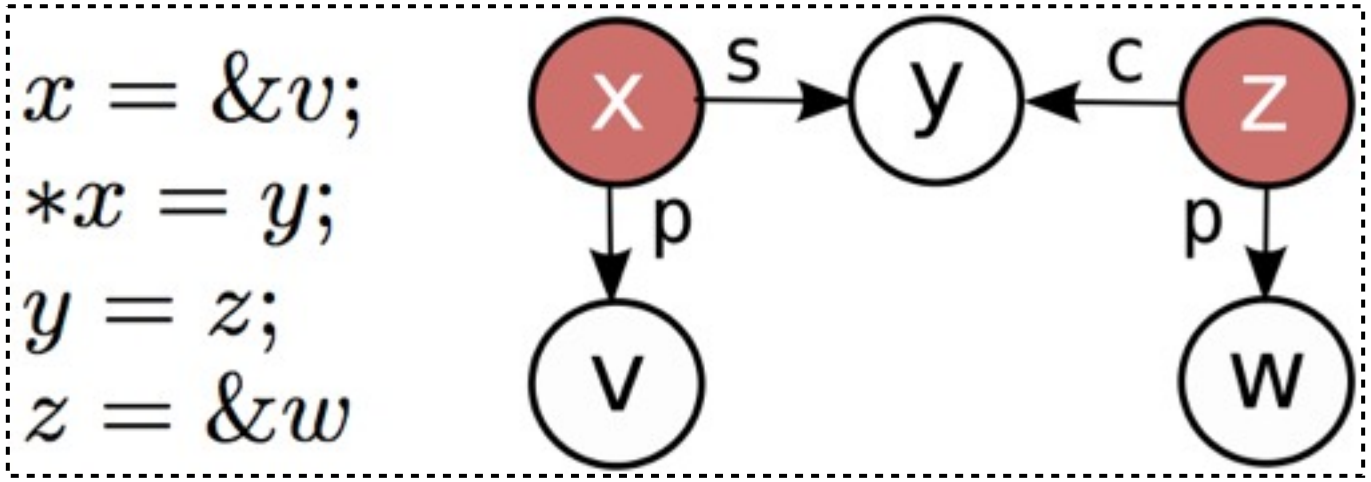
1. **Initialization**
   read statements related to pointer manipulations

2. **Constraint graph creation**
   For each pointer variable in the input program, add a new node to a constraint graph



Code	Name	Edge
x=&y	points	$x \xrightarrow{p} y$
x=y	copy	$x \xleftarrow{c} y$
x=*y	load	$x \xleftarrow{l} y$
*x=y	store	$x \xrightarrow{s} y$
x=y+o	addPtr	$x \xleftarrow{a,o} y$

Basic Edge types



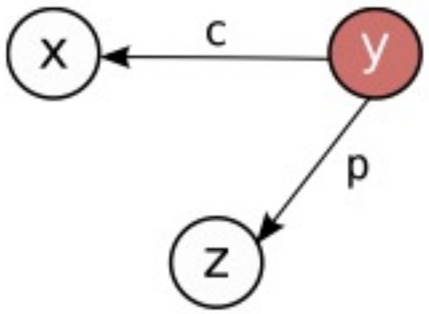
$x = \&v;$
$*x = y;$
$y = z;$
$z = \&w$

Ex.) graph creation

Step 3 : Solving constraints

copy

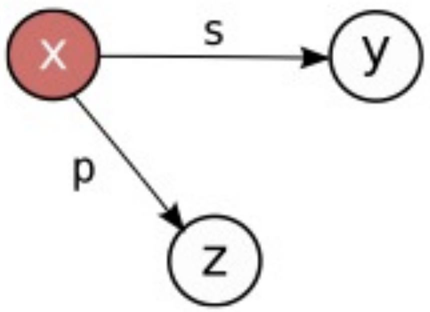
$y \xrightarrow{p} z \wedge y \xrightarrow{c} x \Rightarrow x \xrightarrow{p} z$



$pts(x) \supseteq pts(y)$

store

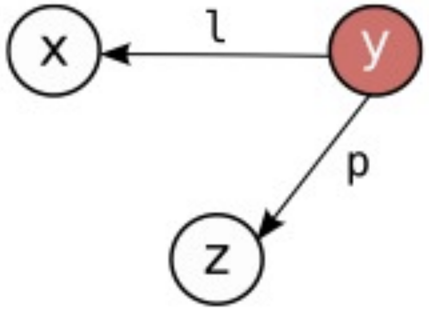
$x \xrightarrow{p} z \wedge x \xrightarrow{s} y \Rightarrow y \xrightarrow{c} z$



$\forall z \in pts(x) : pts(z) \supseteq pts(y)$

load

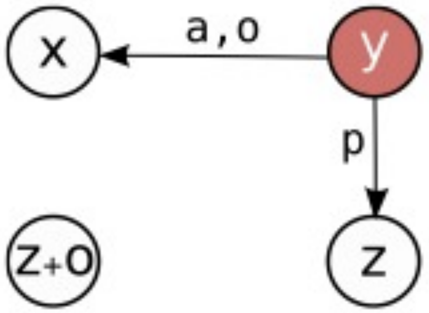
$y \xrightarrow{p} z \wedge y \xrightarrow{l} x \Rightarrow z \xrightarrow{c} x$



$\forall z \in pts(y) : pts(x) \supseteq pts(z)$

addPtr

$y \xrightarrow{p} z \wedge y \xrightarrow{a,o} x \Rightarrow x \xrightarrow{p} z + o$

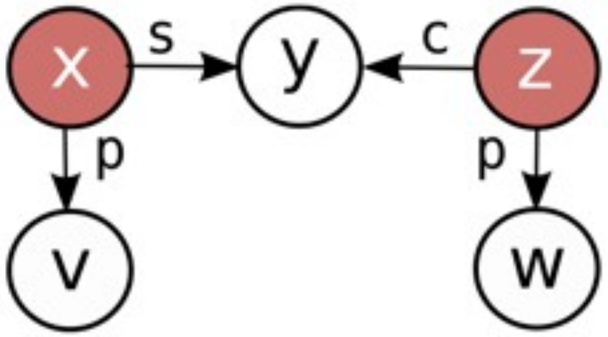


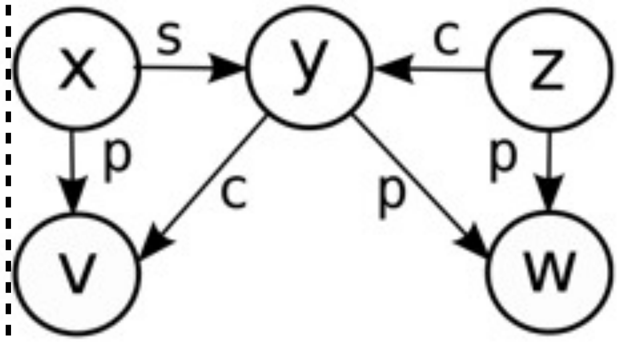
$pts(x) \supseteq \{ z + o \mid z \in pts(y)\}$

Constraint Graph rewriting rules

Example

$x = \&v;$
$*x = y;$
$y = z;$
$z = \&w$





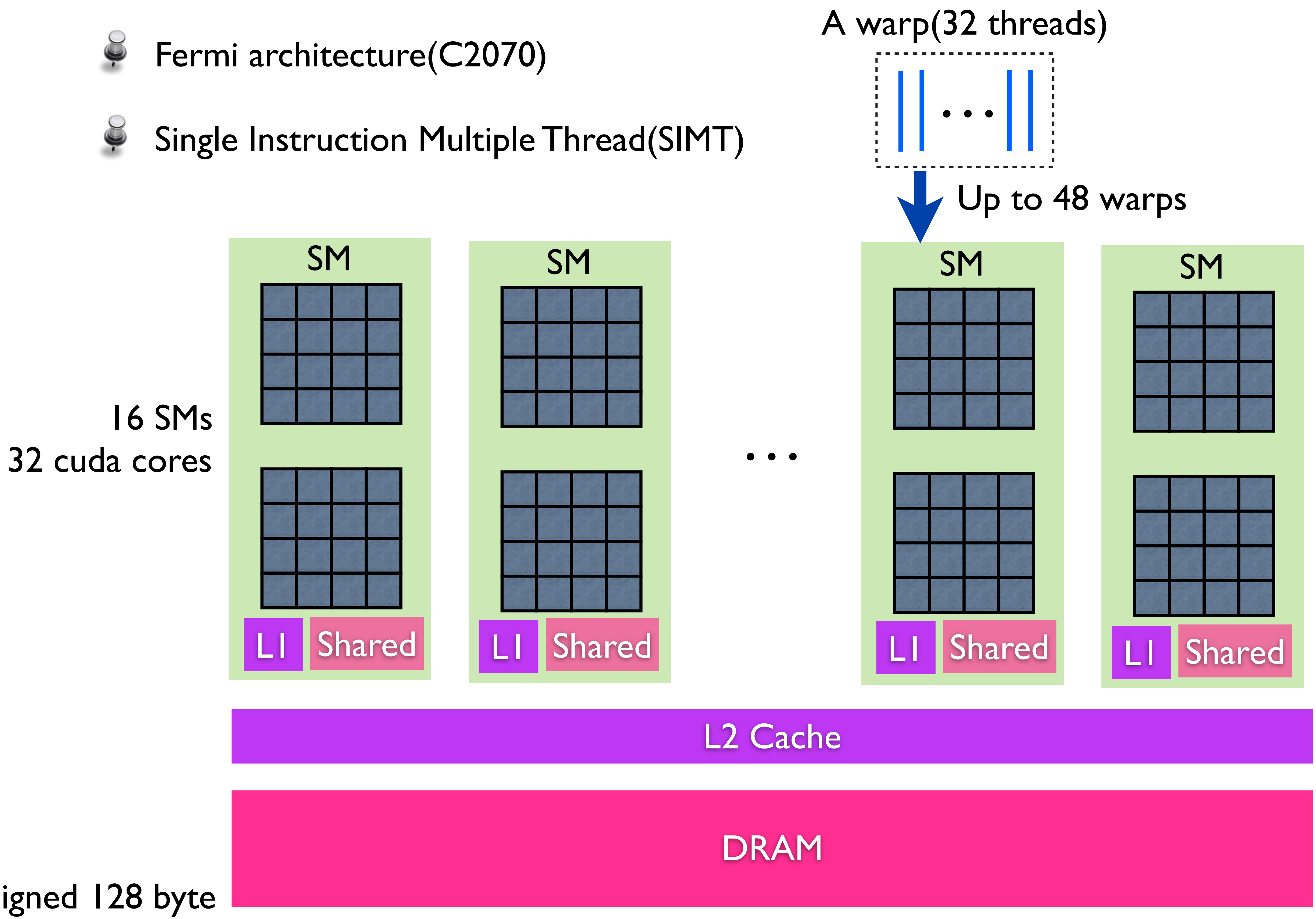
solution

var	pts
x	$\{v\}$
y	$\{w\}$
z	$\{w\}$
v	$\{w\}$
w	$\{\}$

3. GPU architecture and programming model

3.1 GPU architecture and programming model

- Fermi architecture(C2070)
- Single Instruction Multiple Thread(SIMT)



A warp(32 threads)

Up to 48 warps

16 SMs
32 cuda cores

SM

L1 Shared

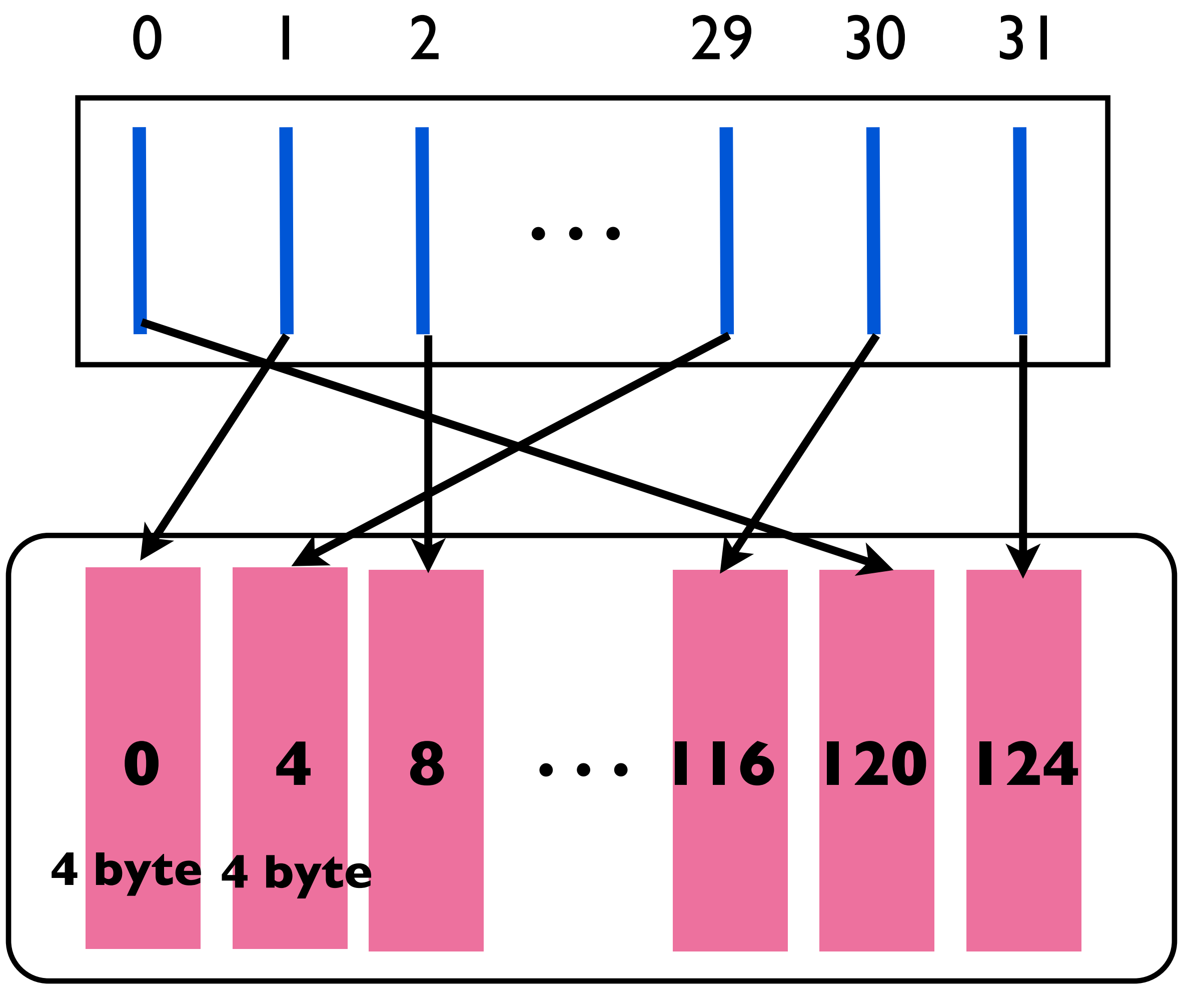
L2 Cache

DRAM

Aligned 128 byte
The hardware merges the 32 reads or writes into one coalesced memory transaction

※ Bank Conflict

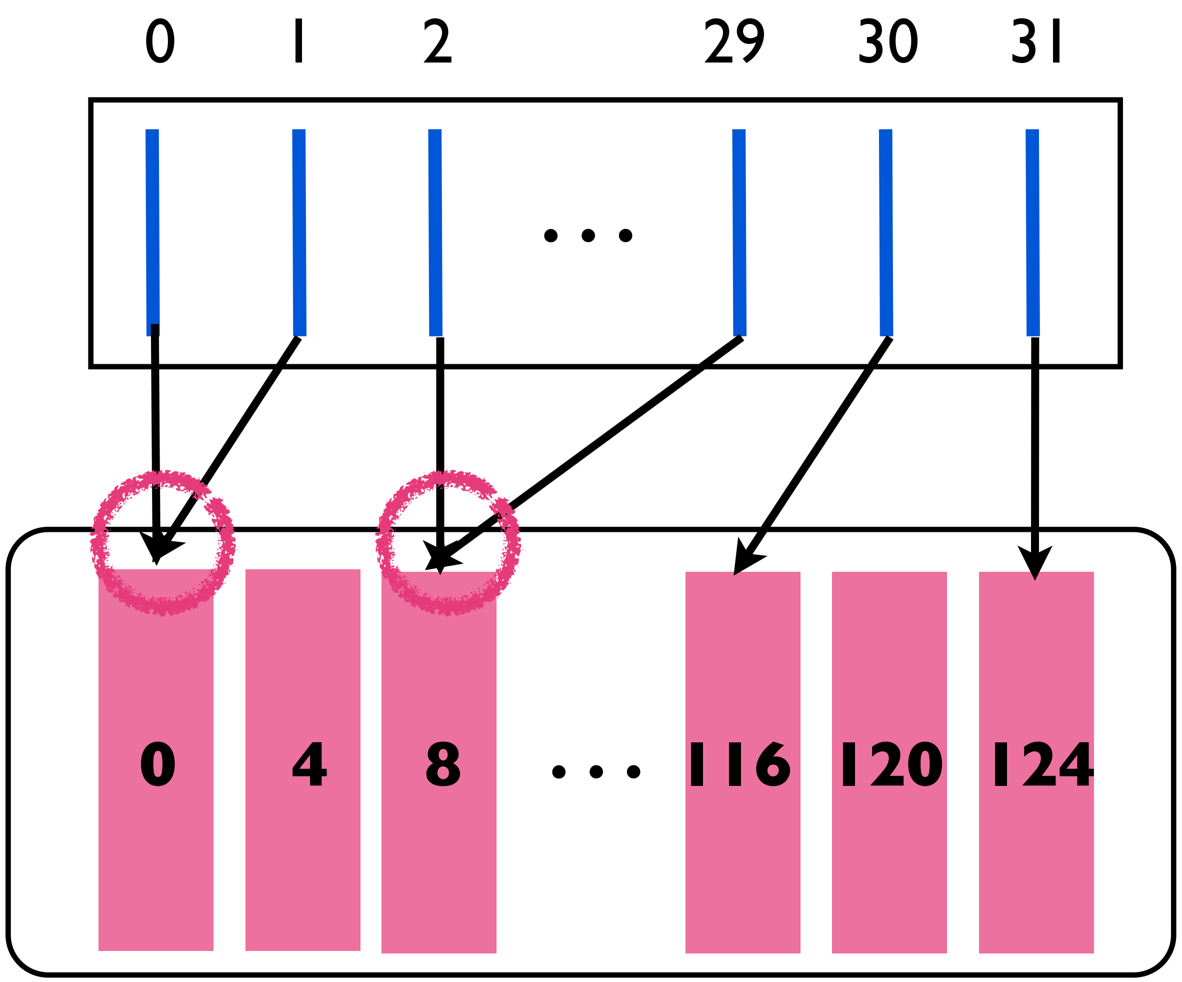
Threads(a warp)



Shared memory banks
1word(4byte) * 32

No Bank Conflict

Threads(a warp)



Shared memory banks
1word(4byte) * 32

2-way Bank Conflict

4. Graph representation on the GPU

4.1 Graph representation on the GPU

- The analysis of the linux kernel results in a constraint graph with 1.498 billion edges

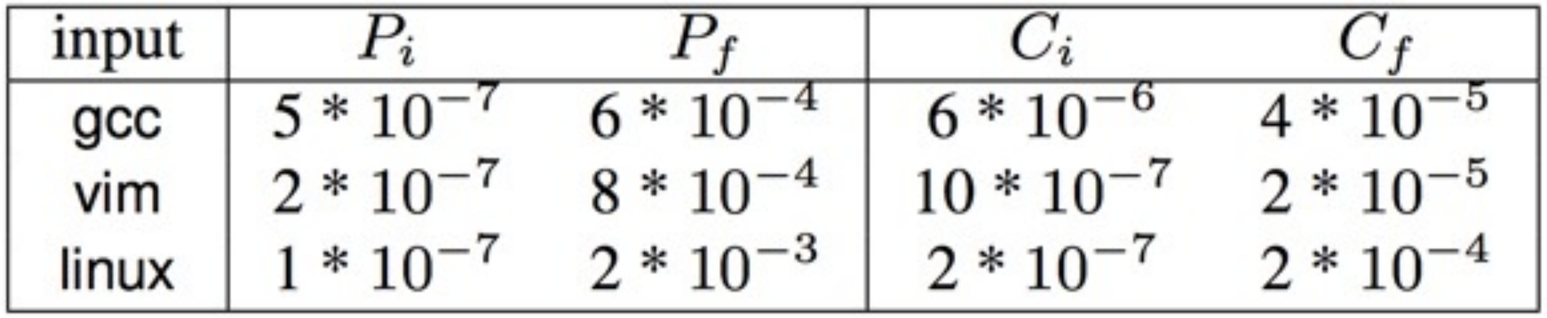
- The memory layout of the graph has to be specifically designed for the GPU architecture to...
  - ✓ minimize memory transactions
  - ✓ maximize coalescing
  - ✓ avoid divergence within the threads of a warp

4.2 adjacency matrix model

- ***n*** **× *n* dense matrix**
  where *n* is the number of the variables

○ **Graph rewrite rules -> matrix-matrix multiplications be performed quickly(CUBLUS)**

× **Wast a lot of space**



input	P_i	P_f	C_i	C_f
gcc	$5 * 10^{-7}$	$6 * 10^{-4}$	$6 * 10^{-6}$	$4 * 10^{-5}$
vim	$2 * 10^{-7}$	$8 * 10^{-4}$	$10 * 10^{-7}$	$2 * 10^{-5}$
linux	$1 * 10^{-7}$	$2 * 10^{-3}$	$2 * 10^{-7}$	$2 * 10^{-4}$

4.2 CSR model

- ***CSR : Compressed Sparse Row***
  where *n* is the number of the variables

○ **Can use space efficiently**

✕ **Difficult to add edges dynamically**

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

col_ind = {1,2,4,1,3}

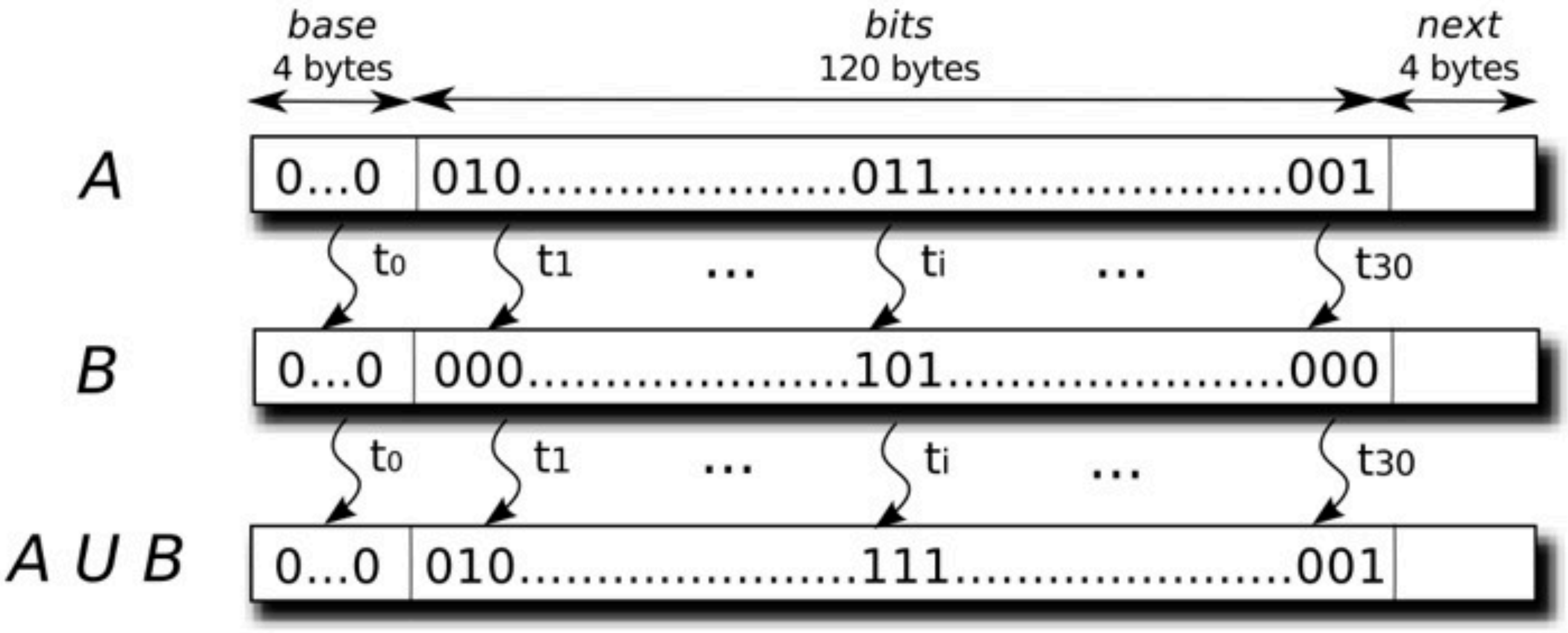
row_ptr = {1,2,4}

4.2 Sparse bit vector model - 1/2

- Linked list, three fields (base, bit, next-ptr)

- 128 bytes width matches the GPU memory bus
  128 bytes / 32 threads(warp) = 4 byte/threads

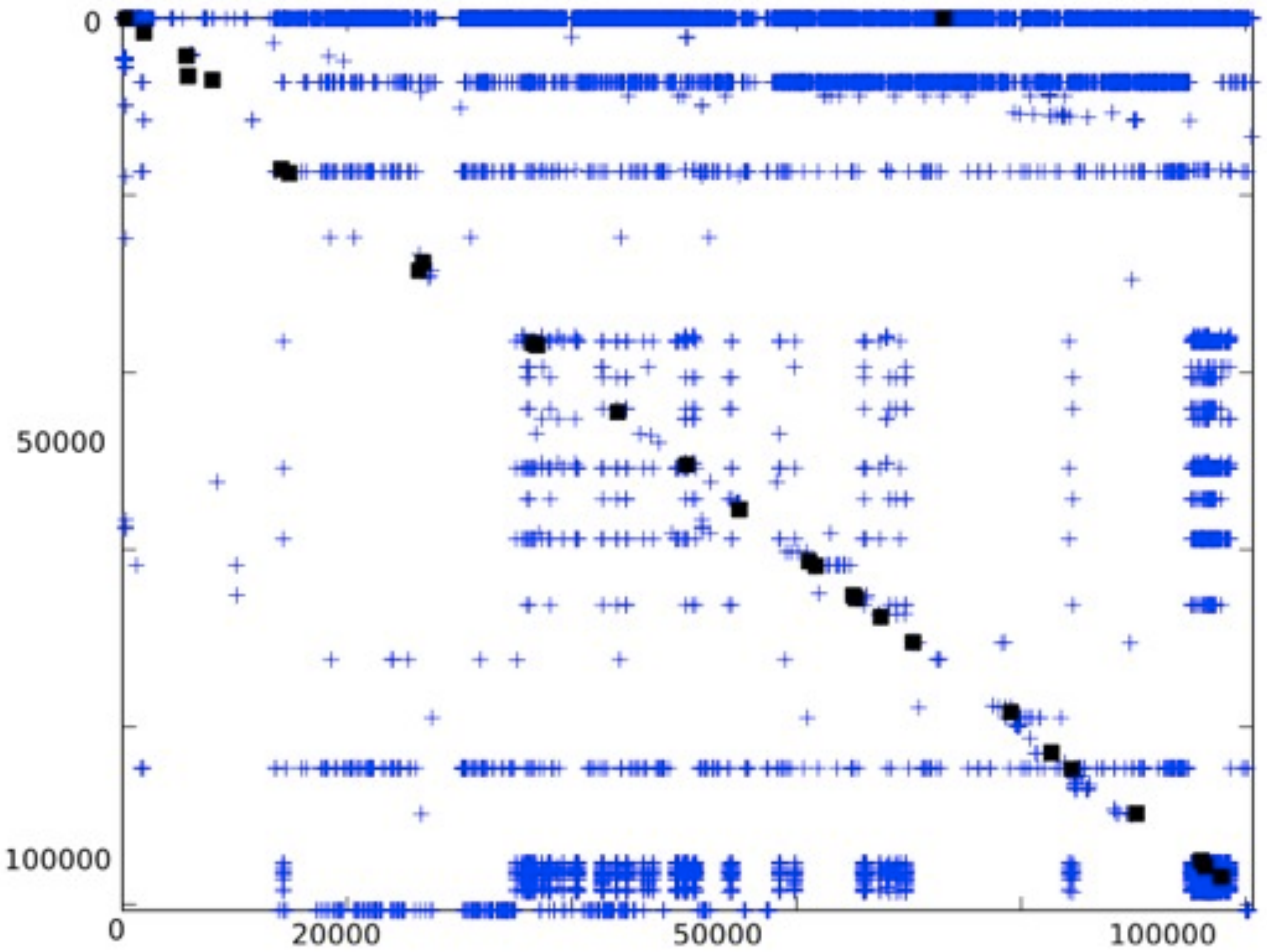
- Is 120-byte large size?(in many CPU implementations is 4 bytes)
  960 elements occupies 120 bytes
  while the standard representation requires 360 bytes (thirty elements)



4.2 Sparse bit vector model - 2/2

- **Identifiers are sequentially assigned to variables as they appear in the program**
  → points to variables with identifiers close to it

- **variables point to others that appear close together**



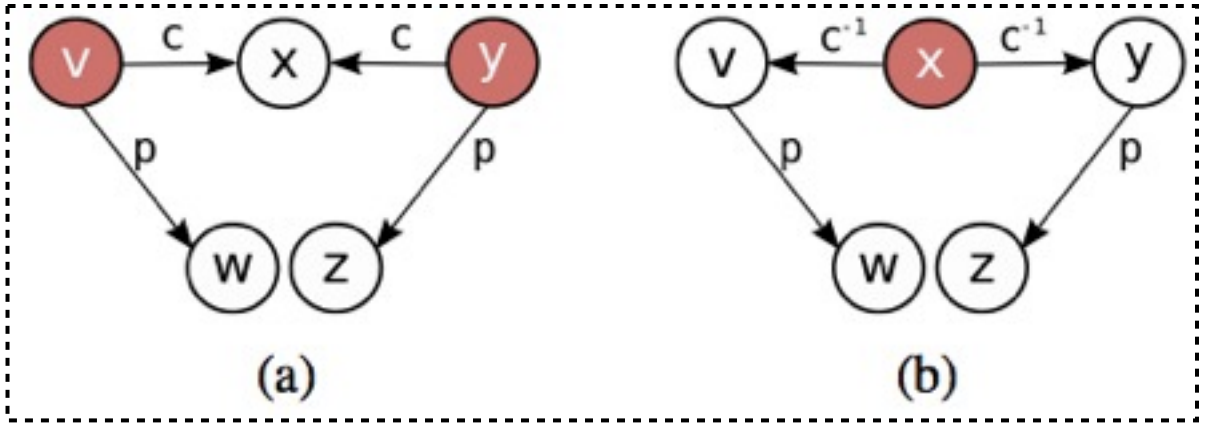
Point-to edges at...
■ at the beginning
+ at the end

Adjacency matrix of the gcc points-to graph(gcc:120K variables)

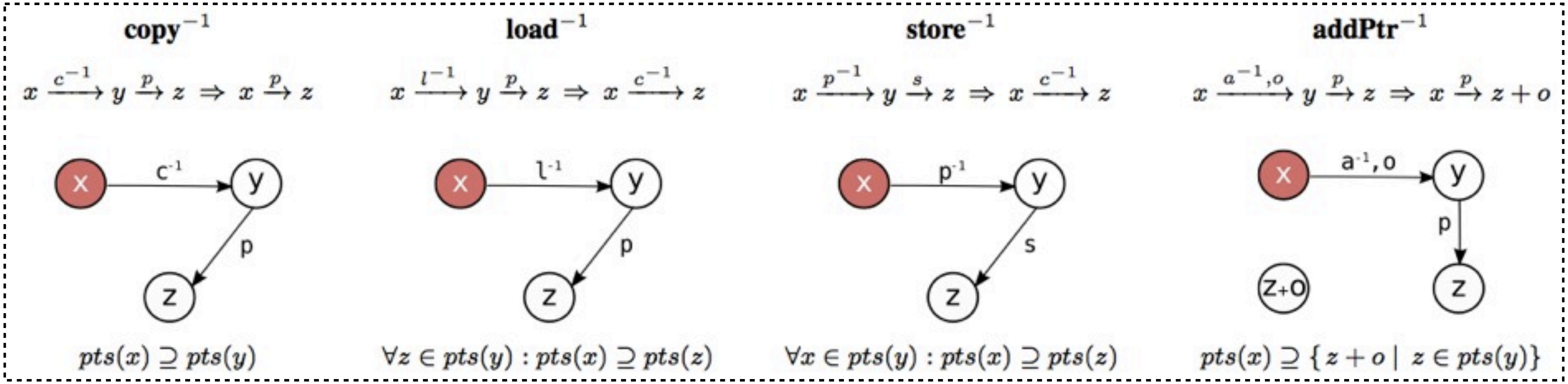
5. Parallel rule application on the GPU

5.1 modify rewrite rule

- **Reduce synchronization**

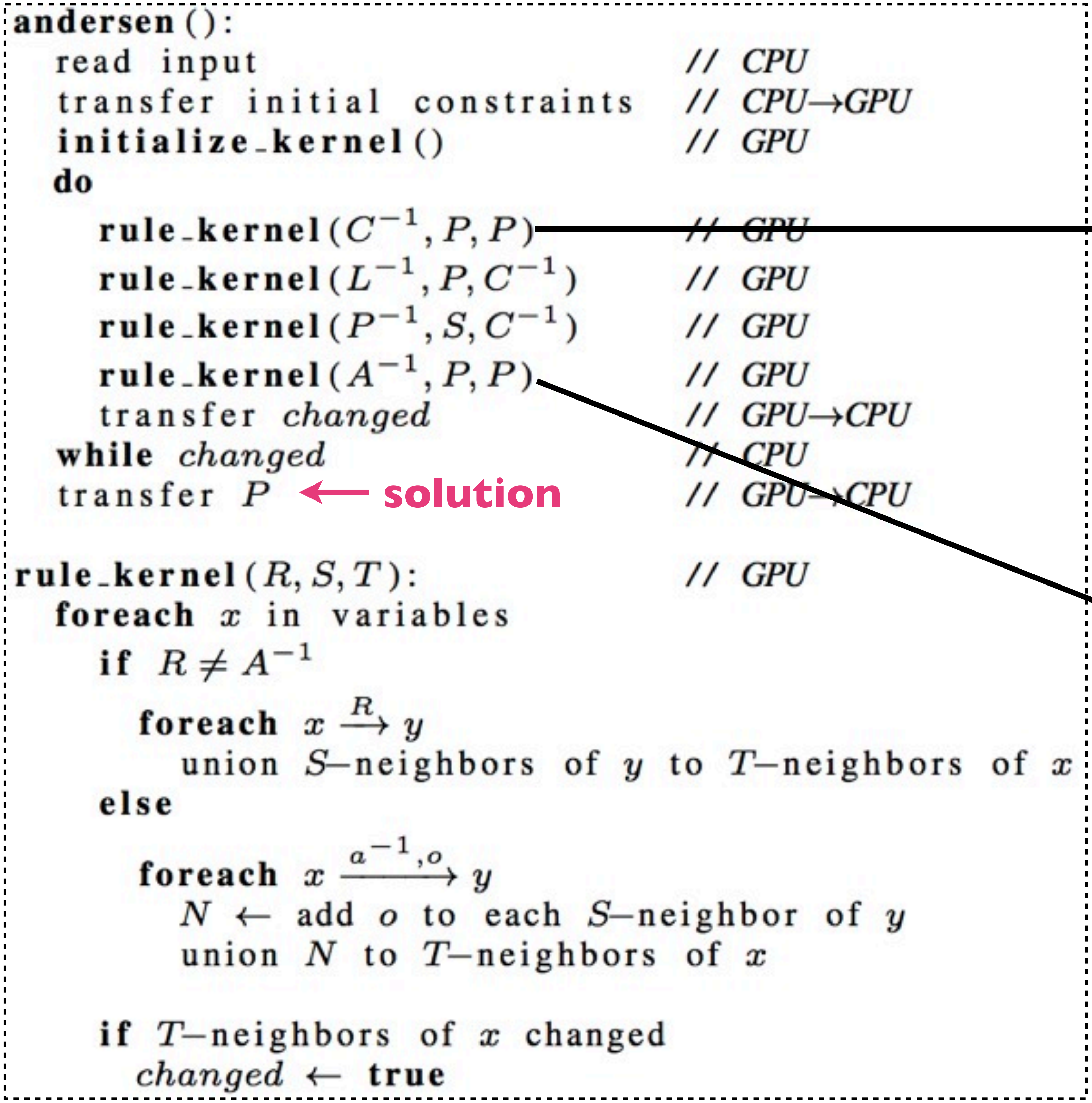


Normal rule Reversed rule


$\mathbf{copy}^{-1}$
$x \xrightarrow{c^{-1}} y \xrightarrow{p} z \Rightarrow x \xrightarrow{p} z$
x
c^{-1}
y
p
z
$pts(x) \supseteq pts(y)$
$\mathbf{load}^{-1}$
$x \xrightarrow{l^{-1}} y \xrightarrow{p} z \Rightarrow x \xrightarrow{c^{-1}} z$
x
l^{-1}
y
p
z
$\forall z \in pts(y) : pts(x) \supseteq pts(z)$
$\mathbf{store}^{-1}$
$x \xrightarrow{p^{-1}} y \xrightarrow{s} z \Rightarrow x \xrightarrow{c^{-1}} z$
x
p^{-1}
y
s
z
$\forall x \in pts(y) : pts(x) \supseteq pts(z)$
$\mathbf{addPtr}^{-1}$
$x \xrightarrow{a^{-1},o} y \xrightarrow{p} z \Rightarrow x \xrightarrow{p} z + o$
x
a^{-1},o
y
p
z+o
z
$pts(x) \supseteq \{ z + o \mid z \in pts(y) \}$

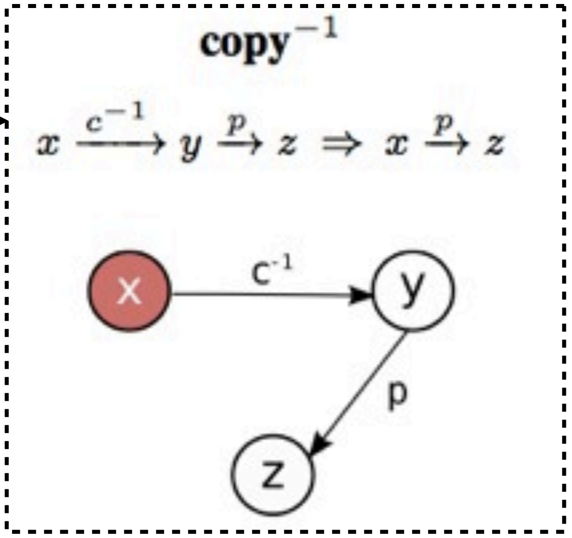
Reversed rule

5.1 Pseudo-code(GPU)



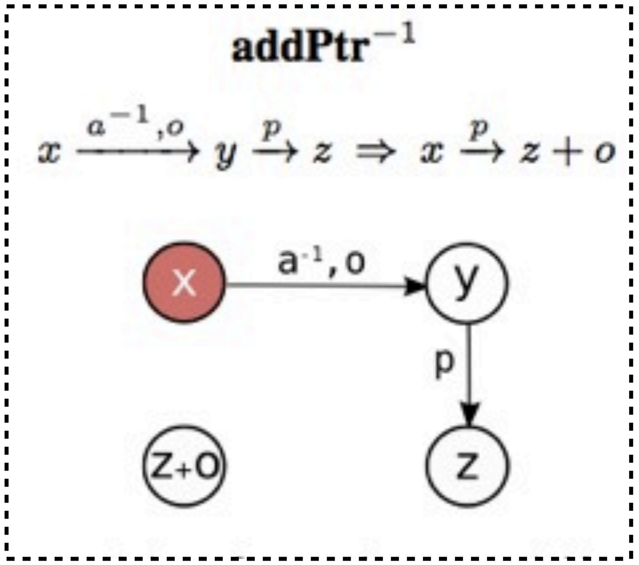
```
andersen():
  read input                                   // CPU
  transfer initial constraints                 // CPU→GPU
  initialize_kernel()                          // GPU
  do
    rule_kernel($C^{-1}, P, P$)                // GPU
    rule_kernel($L^{-1}, P, C^{-1}$)           // GPU
    rule_kernel($P^{-1}, S, C^{-1}$)           // GPU
    rule_kernel($A^{-1}, P, P$)                // GPU
    transfer changed                           // GPU→CPU
  while changed                                // CPU
  transfer P  ⟵ solution                       // GPU→CPU

rule_kernel($R, S, T$):                        // GPU
  foreach $x$ in variables
    if $R \neq A^{-1}$
      foreach $x \xrightarrow{R} y$
        union $S$–neighbors of $y$ to $T$–neighbors of $x$
    else
      foreach $x \xrightarrow{a^{-1},o} y$
        $N \leftarrow$ add $o$ to each $S$–neighbor of $y$
        union $N$ to $T$–neighbors of $x$

    if $T$–neighbors of $x$ changed
      changed ← true
```

Example.



copy^{-1}

$x \xrightarrow{c^{-1}} y \xrightarrow{p} z \Rightarrow x \xrightarrow{p} z$



addPtr^{-1}

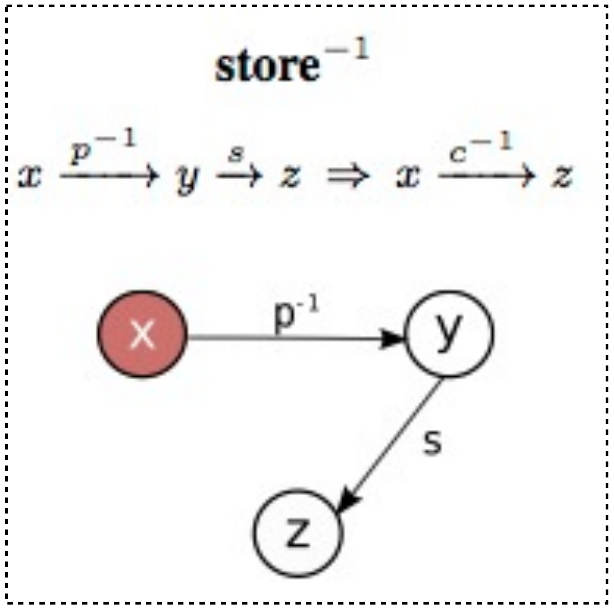
$x \xrightarrow{a^{-1},o} y \xrightarrow{p} z \Rightarrow x \xrightarrow{p} z+o$

6. Optimizations

6. Optimizations 1/2

1) Minimize memory consumption
use pair list (x, y) , instead of storing p-1

edges data
※ y has outgoing “s” edges and y → x


store^{-1}
$x \xrightarrow{p^{-1}} y \xrightarrow{s} z \Rightarrow x \xrightarrow{c^{-1}} z$
x
p-1
y
s
z

2) Collapse cycles detection
Offline : look for cycles during a preprocessing phase
a = b; b = a;
Online : look for cycles during the solving process
***a = b; b = *a;**

HCD(Hybrid Cycle Detection)
combine the Offline and Online method

※ Offline method implementation is only CPU ver.

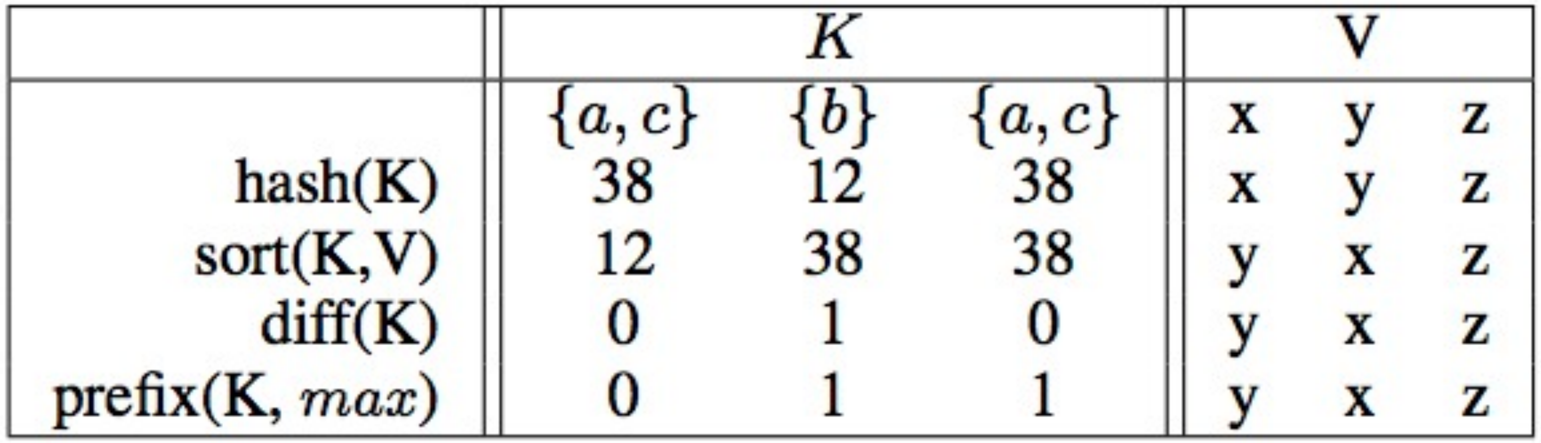
6. Optimizations 2/2

3) Avoid redundant rule application

- **If the points-to sets of all the variables have not changed at current iteration, then return solution**
- **transfer ΔP from the GPU to CPU by using streams**
  ΔP : edges which added during the last and current iterations
  ΔP is updated in the end of each iteration.
  Δ*P* = Δ*P* - *P*   and *P* = *P U* Δ*P*
- **Computing differences between sets of edges is suited for warp centric model**

4) Detect pointer-equivalent variables

ΔP-equivalent variables have the same outgoing ΔP edges in the current iteration



	K			V		
	$\{a, c\}$	$\{b\}$	$\{a, c\}$	x	y	z
hash(K)	38	12	38	x	y	z
sort(K,V)	12	38	38	y	x	z
diff(K)	0	1	0	y	x	z
prefix(K, max)	0	1	1	y	x	z

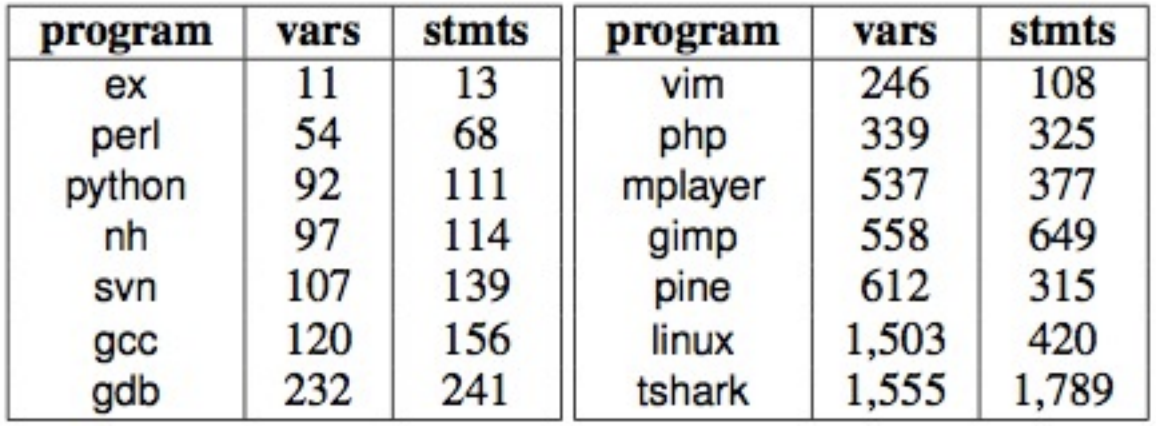
Example of detection of ΔP-equivalent variables

7. Experimental evaluation

7.1 Experimental evaluation

Three implementation..

1. **CPU**
2. **Multi-CPU**
3. **GPU**



program	vars	stmts	program	vars	stmts
ex	11	13	vim	246	108
perl	54	68	php	339	325
python	92	111	mplayer	537	377
nh	97	114	gimp	558	649
svn	107	139	pine	612	315
gcc	120	156	linux	1,503	420
gdb	232	241	tshark	1,555	1,789

Benchmark suite: number of variables and statements (in thousands)

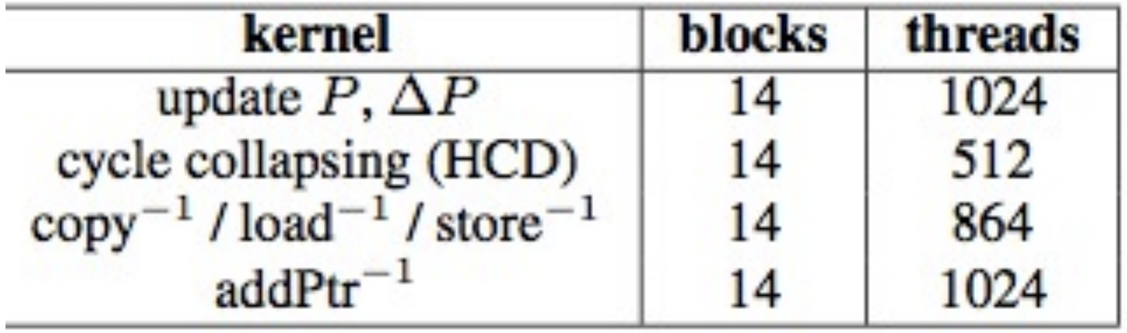
7.2 Experiment environment

1. CPU

   AMD Opteron 4-core 2.7GHz * 4 Socket
   Ubuntu 10
   Memory : 24GB
   L1 : 64KB, L2 : 512KB, L3 : 6MB
   C++ , -O3
   Multi-CPU implementation is written in Java
   JVM : 64-bit un HotSpot server version 1.6.0 24.

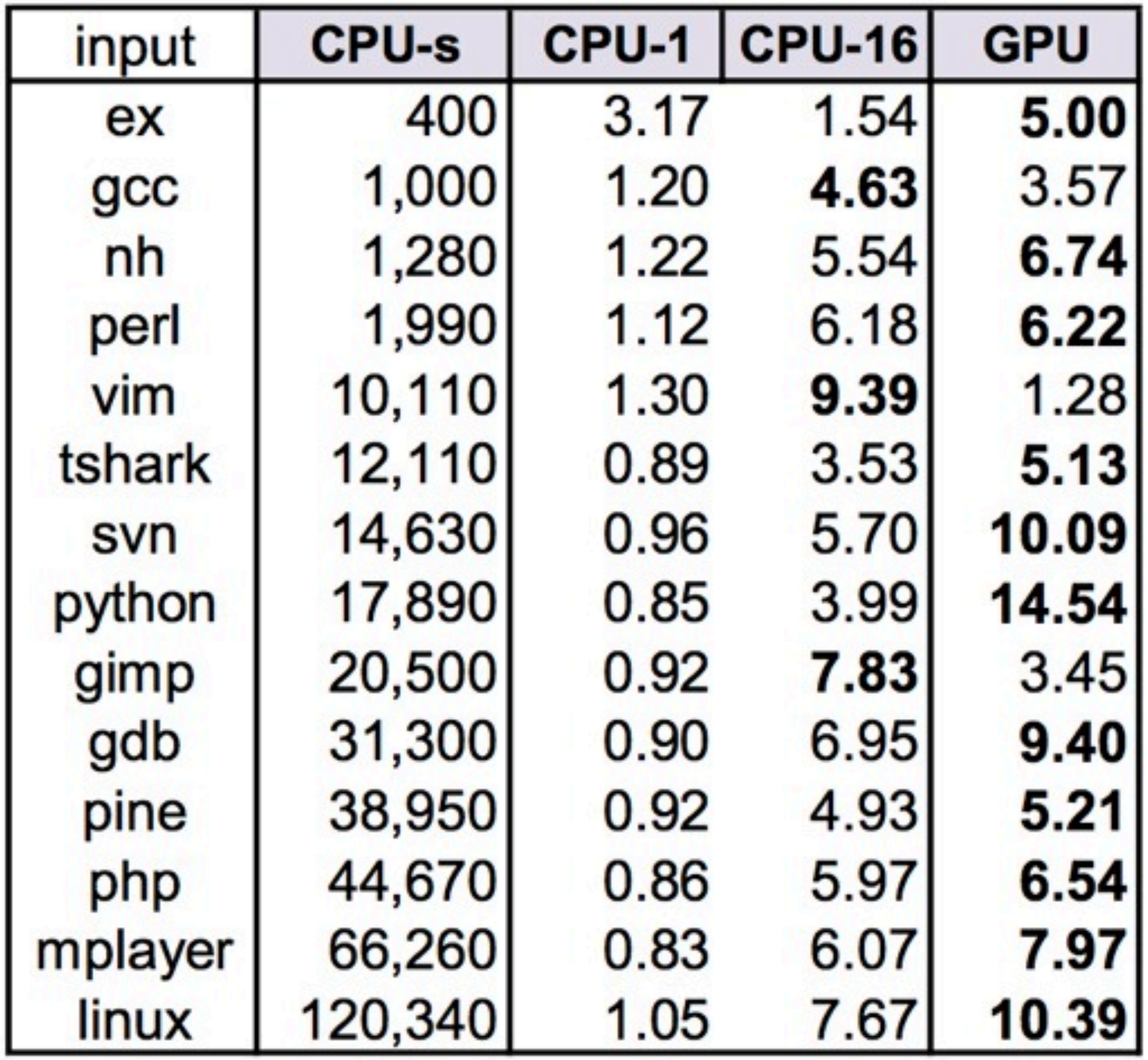
2. GPU

   NVIDIA Tesla C2070(1.15 GHz)
   14 SMs, 448 cuda cores
   Memory : 6GB
   L1 : 16 KB, L2 : 768 KB
   Shared memory : 48 KB
   CUDA 4.1



kernel	blocks	threads
update P, ΔP	14	1024
cycle collapsing (HCD)	14	512
copy^{-1} / load^{-1} / store^{-1}	14	864
addPtr^{-1}	14	1024

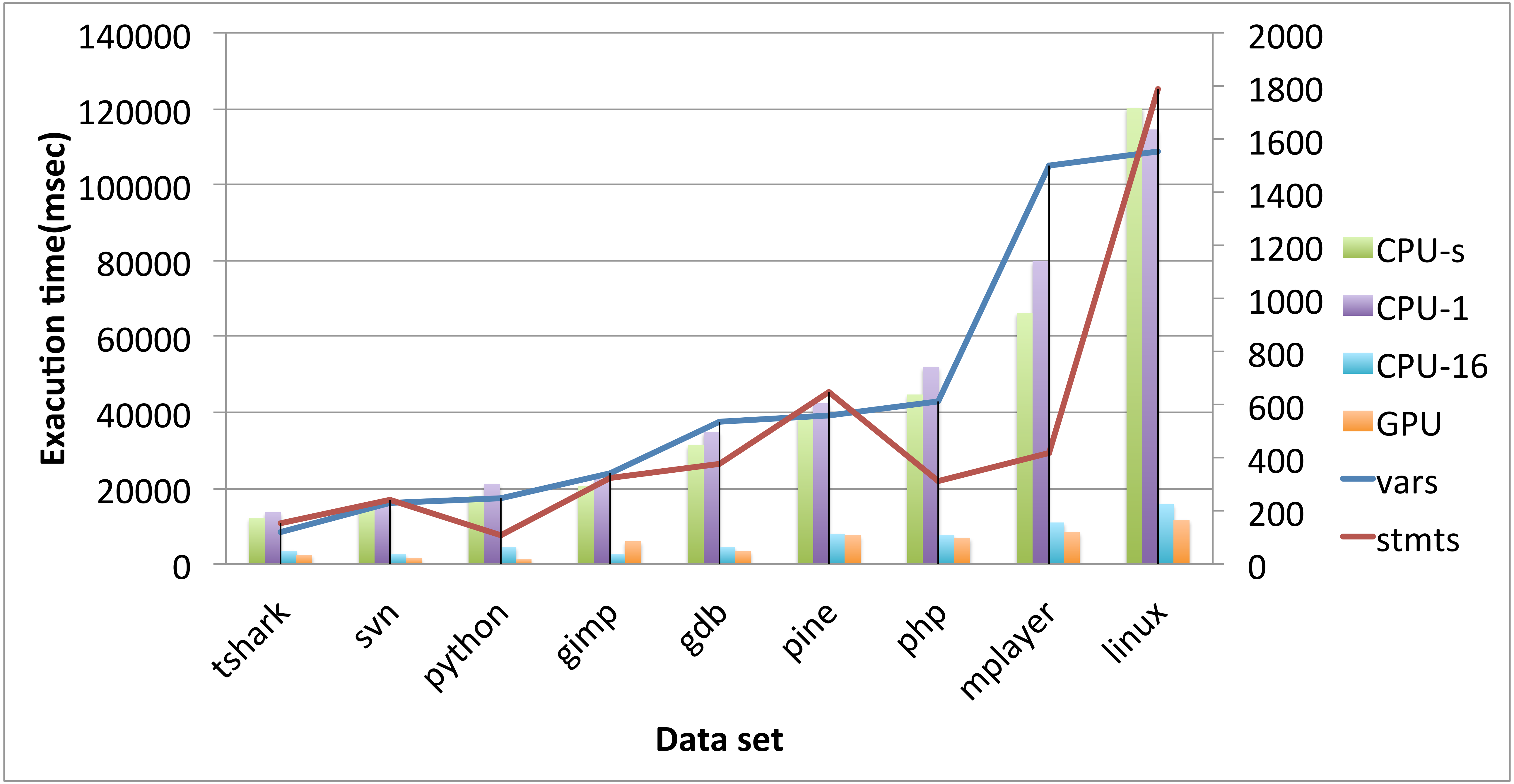
7.2 Experiment result



input	CPU-s	CPU-1	CPU-16	GPU
ex	400	3.17	1.54	**5.00**
gcc	1,000	1.20	**4.63**	3.57
nh	1,280	1.22	5.54	**6.74**
perl	1,990	1.12	6.18	**6.22**
vim	10,110	1.30	**9.39**	1.28
tshark	12,110	0.89	3.53	**5.13**
svn	14,630	0.96	5.70	**10.09**
python	17,890	0.85	3.99	**14.54**
gimp	20,500	0.92	**7.83**	3.45
gdb	31,300	0.90	6.95	**9.40**
pine	38,950	0.92	4.93	**5.21**
php	44,670	0.86	5.97	**6.54**
mplayer	66,260	0.83	6.07	**7.97**
linux	120,340	1.05	7.67	**10.39**

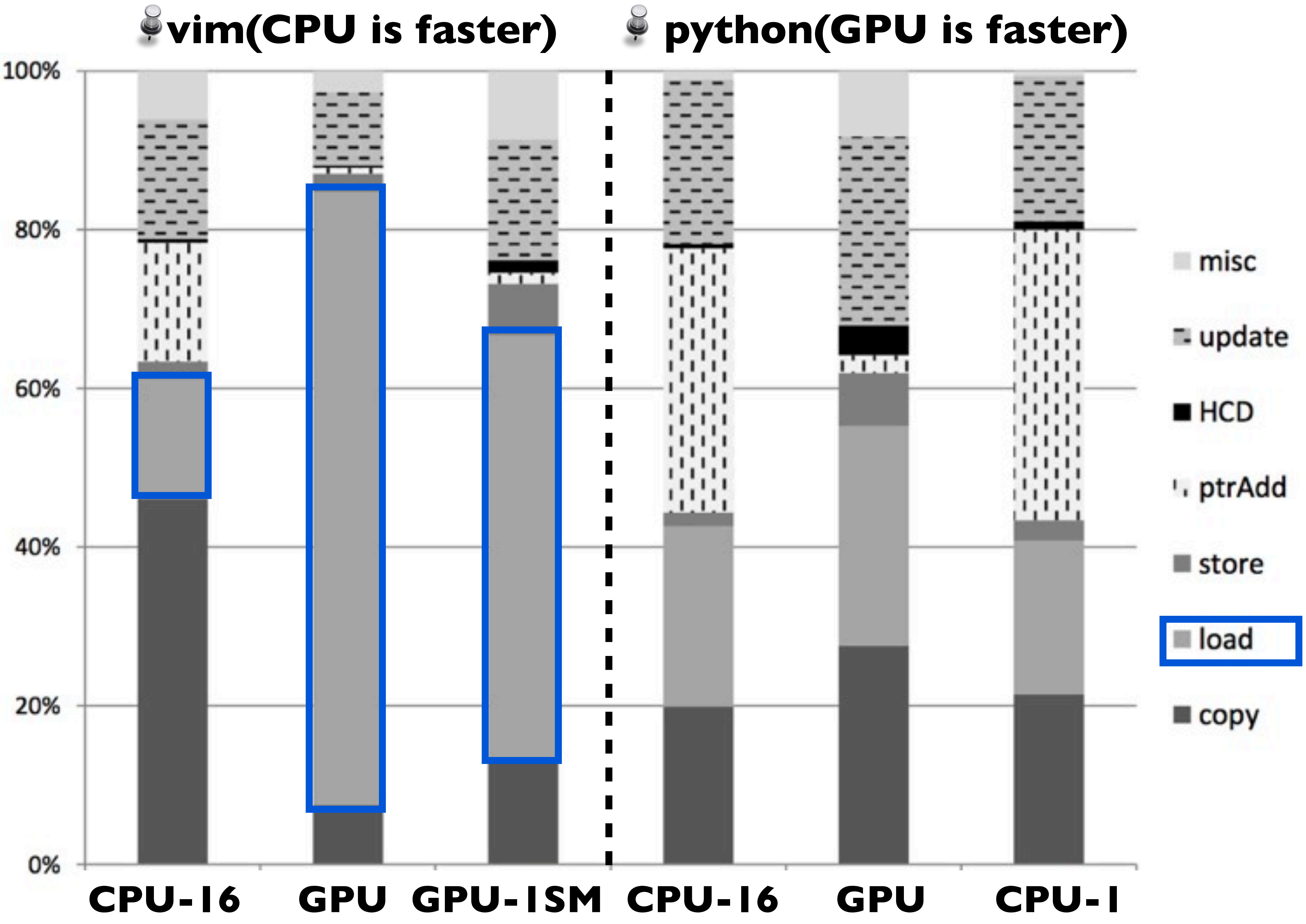
Runtimes(in ms)for the sequential online phase(CPU-s column), and speedups achieved by CPU-x and GPU

7.2 Experiment result



Runtimes(in ms)for the sequential online phase(CPU-s column), and speedups achieved by CPU-x and GPU

7.2 Breakdown of the execution time



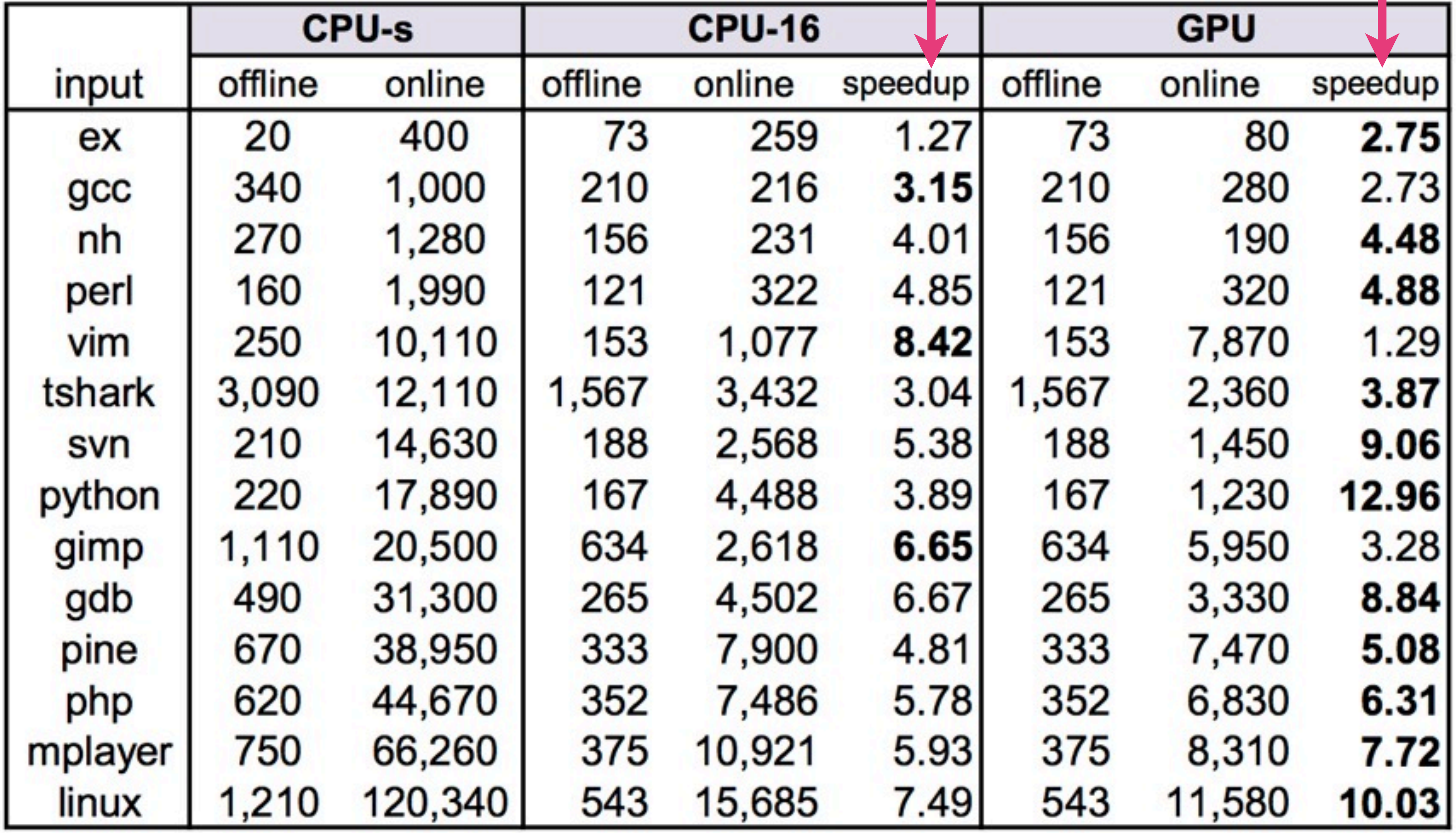
Breakdown of the execution time for the vim and python benchmarks

7.2 Compares the total analysis runtimes

- **Offline phase is always executed on the CPU**
- **Data exchanging time CPU ↔ GPU is not bottleneck (due to overlapping the data transfer)**

Average : 6x

Average : 7x



input	CPU-s		CPU-16			GPU		
	offline	online	offline	online	speedup	offline	online	speedup
ex	20	400	73	259	1.27	73	80	**2.75**
gcc	340	1,000	210	216	**3.15**	210	280	2.73
nh	270	1,280	156	231	4.01	156	190	**4.48**
perl	160	1,990	121	322	4.85	121	320	**4.88**
vim	250	10,110	153	1,077	**8.42**	153	7,870	1.29
tshark	3,090	12,110	1,567	3,432	3.04	1,567	2,360	**3.87**
svn	210	14,630	188	2,568	5.38	188	1,450	**9.06**
python	220	17,890	167	4,488	3.89	167	1,230	**12.96**
gimp	1,110	20,500	634	2,618	**6.65**	634	5,950	3.28
gdb	490	31,300	265	4,502	6.67	265	3,330	**8.84**
pine	670	38,950	333	7,900	4.81	333	7,470	**5.08**
php	620	44,670	352	7,486	5.78	352	6,830	**6.31**
mplayer	750	66,260	375	10,921	5.93	375	8,310	**7.72**
linux	1,210	120,340	543	15,685	7.49	543	11,580	**10.03**

Comparison of runtimes(in ms) for the whole analysis: CPU (sequential), CPU (parallel, 16 threads), and GPU

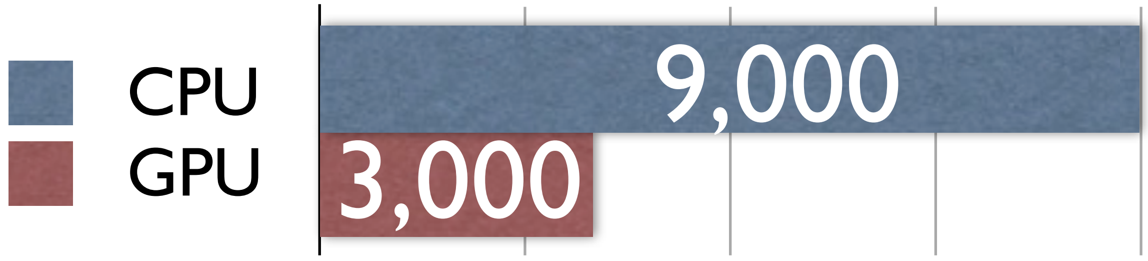
8. Conclusions

8. Conclusions

GPU(14SMs) implementation achieves

- ✓ **7x speedup compare to a sequential CPU ver.**
- ✓ **outperforms a same algorithm on 16 CPU cores**

- **35% more person-hours to implementation on the GPU**
- **GPU(CUDA) version is quite compact**



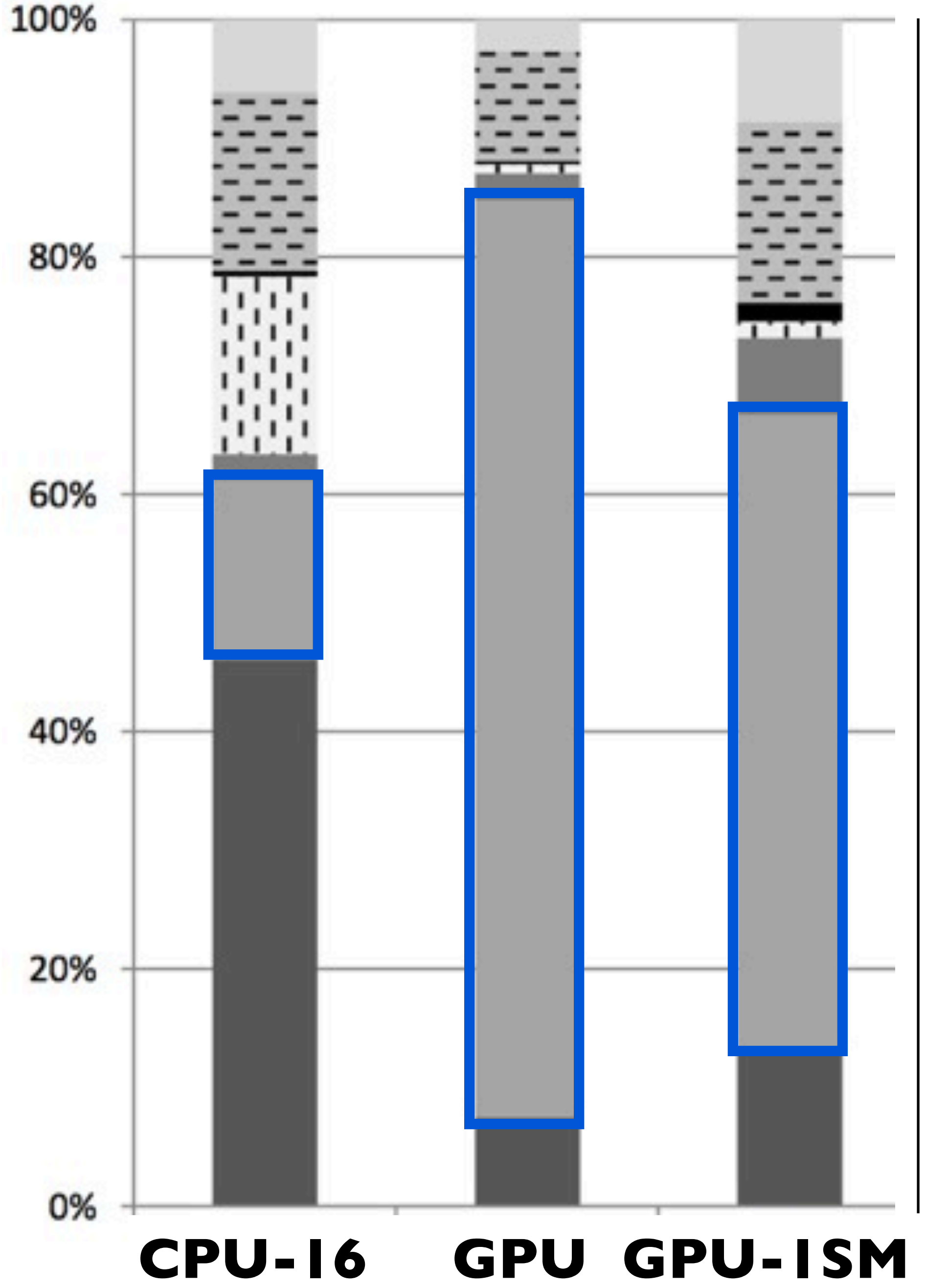
Source code size(line)

※Due to size of Implementation of data structure

Point-to Analysis using BDDs

7.2 Breakdown of the execution time

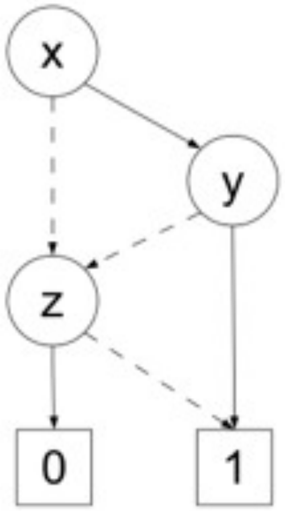
vim(CPU is faster)



Breakdown of the execution time for the vim and python benchmarks

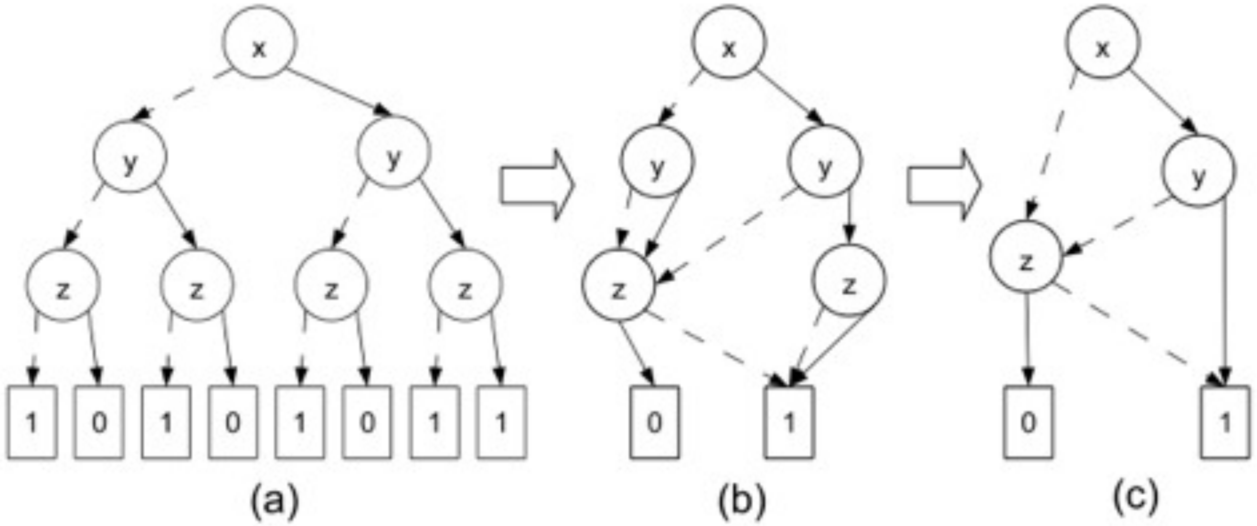
※ Binary Decision Diagram(BDD)

Bool function using BDD



$f(x, y, z) = x * y + \neg z$

Reduced BDD

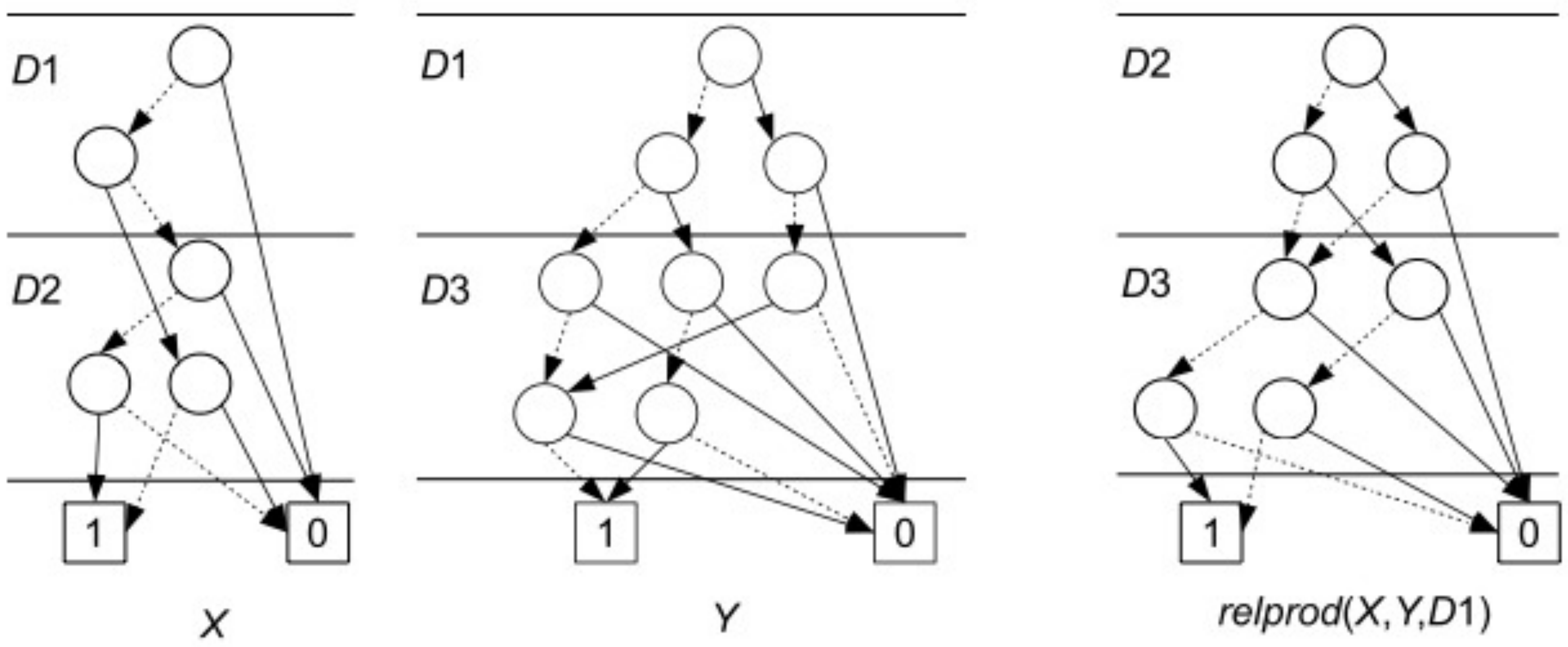


○ **Efficient memory space & Low calculation cost(in proportion to size of the graph)**

✕ **Finding the best variable ordering is NP-hard**

※ Point-to Analysis using BDDs

X : D1×D2, X = {(00,01), (01,00), (01,10)}
Y : D1×D3, X = {(00,00), (01,01), (10,10)}
R : X & Y = {(00,01), (01,00), (10,01)}



Point-to **Copy-to** **Point-to**

Memoizing
Reuse previously processed inputs to reduce redundant work